# Rapid Feature Learning with Stacked Linear Denoisers

**Zhixiang Eddie Xu**
Department of Computer Science
Washington University in St. Louis
Saint Louis, MO 63108
xuzx@cse.wustl.edu

**Kilian Q. Weinberger**
Department of Computer Science
Washington University in St. Louis
Saint Louis, MO 63108
kilian@wustl.edu

**Fei Sha**
Computer Science Department
University of Southern California
Los Angeles, CA 90089
feisha@usc.edu

## Abstract

We investigate unsupervised pre-training of deep architectures as feature generators for "shallow" classifiers. Stacked Denoising Autoencoders (SdA) [23], when used as feature pre-processing tools for SVM classification, can lead to significant improvements in accuracy – however, at the price of a substantial increase in computational cost. In this paper we create a simple algorithm which mimics the layer by layer training of SdAs. However, in contrast to SdAs, our algorithm requires no training through gradient descent as the parameters can be computed in closed-form. It can be implemented in less than 20 lines of MATLAB$^{\text{TM}}$and reduces the computation time from several hours to mere seconds. We show that our feature transformation reliably improves the results of SVM classification significantly on all our data sets – often outperforming SdAs and even deep neural networks in three out of four deep learning benchmarks.

## 1 Introduction

Recently, there has been a great deal of attention on "deep-learning" architectures [3, 10]. Such architectures have consistently achieved state-of-the-art results on many challenging learning tasks, including object recognition [19], natural language processing [7], dimensionality reduction [10]. A typical paradigm of deep-learning is to first perform unsupervised pre-training on the neural network to initialize the weights and then use back-propagation for supervised training.

[17] showed that the second phase, the supervised back-propagation, can be replaced with "shallow" classifiers, such as support vector machines (SVM) [8]. Concretely, the outputs of the hidden units of pre-trained deep neural networks are used as input features to those classifiers. Besides achieving superior performance in recognition tasks, the substituting classifiers offer appealing computational properties. In particular, their parameters are adjusted with convex optimization, free of local optima that often plague back-propagation based techniques. Further, these classifiers tend to be more ready to be used out-of-the-box and can often be parallelized very effectively.

*How can we replace the pre-training phase with an equally attractive learning model*? Note that the pre-training phrase – now merely used for unsupervised feature generation – is a major bottleneck in applying deep learning architectures. One needs to adjust several parameters of network architectures (the number of hidden layers, units in each hidden layer), optimization (learning rates

and momentum), etc. Compounded by these factors, the pre-training phase takes up a large portion of the overall training time (anecdotally 50-90%) – even with the use of multi-core processors and graphical processing units (GPUs) [1].

We propose such a method for pre-training, thus for feature generation, and investigate its effectiveness in this paper. We show how one-layer *linear* denoising autoencoders can be used as the basic building blocks of the pre-training phases. The autoencoders are ordinary linear regression models for reconstructing data from corrupted features. They have closed-form solutions and thus are easy to implement. In particular, the parameters can be identified with matrix inversion and nonlinear optimization is not needed. We use the outputs of linear denoising autoencoders in two ways: i) as input features to support vector machines; ii) as inputs (to be denoised) to successively stacked linear autoencoders.

The proposed method, which we refer to as *Stacked Linear Denoiser* (SLIDE), is similar in spirit to stacked (nonlinear) denoising autoencoders (SdA) [23]. However, there are important differences. First, in SdA, hidden layers are used to extract new representations from inputs which makes their training a nonconvex optimization. In contrast, SLIDE does not have hidden layers and is convex. Second, SdA requires the setting of several crucial meta-parameters, including noise level, learning rates, number of training epochs and network architecture specifics, which are typically set by cross-validation. In comparison, SLIDE only has *two* free meta-parameters, controlling the amount of noise to be added to data as well as the number of autoencoders we would like to stack. Finally, leveraging on the analytic tractability of linear regression, we train the parameters of our autoencoders to optimally denoise *all possible* corrupted training inputs — arguably "*infinitely many*". This is practically feasible for SdA, whose parameters are adjusted on only a subset of corrupted data.

In this paper, we make several contributions. SLIDE can be implemented easily (less than 20 lines of Matlab codes). The learning algorithm runs very fast. In fact, our 19-lines implementation is *three orders of magnitudes* faster than a highly optimized parallel implementation of SdA on a state-of-the-art GPU. Even for large data sets with tens of thousands of samples the computation takes only a few seconds – achieving thousand-fold speed-up over the pre-training phase with SdA or similar approaches. Finally, in addition to the vastly reduced computation time, we demonstrate on several deep-learning benchmark data sets that SVM classification with SLIDE features tends to be even more accurate than with SdA features or pre-trained deep neural networks with back-propagation.

## 2 Notation and Background

In the following, we introduce notation and algorithms which are used in the rest of the paper. Our training data consist of $n$ input vectors $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \in \mathcal{R}^d$ with corresponding discrete class labels $\{y_1, \ldots, y_n\}$ drawn from an unknown joint distribution $\mathcal{D}$.

**Deep Architecture.** Deep learning algorithms learn hierarchies of hidden layers, where the output of the lower level layer becomes the input of the higher level. "Deep learning" algorithms differ from traditional neural networks in two ways: i) they tend to have more hidden layers (*i.e.* are *deeper*); ii) the supervised training through back-propagation is preceded by an unsupervised pre-training step in which the weights are initialized in a generative manner. The additional layers are believed to provide more powerful learning models. The pre-training makes the training more efficient (it is often performed greedily – layer by layer) and regularizes the optimization so that back-propagation starts near a "good" local minima [9].

**Support Vector Machines** are one of the most popular and reliable out-of-the-box supervised classification algorithms. SVMs [22] are linear classifiers that involve a quadratic minimization problem, which is convex and not plagued by local optima. The maximum margin separation promotes reliably good generalization, and the *kernel-trick* [22] allows SVMs to generate highly non-linear decisions boundaries with low computational overhead. The kernel-trick maps the input vectors $\mathbf{x}_i$ implicitly into a higher (possibly infinite) dimensional feature space using the kernel function $k(\mathbf{x}_i, \mathbf{x}_j)$. Among various such functions, the Radial Basis Function (RBF)-Kernel, is one of the most commonly used kernels. When used with Euclidean distances (also typically referred to as the

Gaussian kernel) it is defined as follows:

$$k(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|_2^2}{\sigma^2}\right), \tag{1}$$

where $\sigma^2$ denotes the RBF-*kernel-width*.

**Stacked Denoising Autoencoder.** One particular method to pre-train the weights of a deep neural network is Stacked Denoising Autoencoders (SdA) [23, 13]. A traditional autoencoder described in [12] maps the input data, or visible features $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \in \mathcal{R}^d$ into hidden representations $\{\mathbf{h}_1, \ldots, \mathbf{h}_n\} \in \mathcal{R}^m$. The hidden representations are then mapped back to reconstructions of the original input $\{\mathbf{x}'_1, \ldots, \mathbf{x}'_n\} \in \mathcal{R}^d$. For both mappings they use the sigmoid function

$$\mathbf{h}_i = \frac{1}{1 + e^{-(\mathbf{W}\mathbf{x}_i + \mathbf{b})}} \quad \text{and} \quad \mathbf{x}'_i = \frac{1}{1 + e^{-(\mathbf{W}^\top \mathbf{h}_i + \mathbf{b}')}}, \tag{2}$$

where the parameters consist of a weight matrix $\mathbf{W} \in \mathcal{R}^{m \times d}$ and two bias vectors $\mathbf{b}, \mathbf{b}' \in \mathcal{R}^m$. The optimal parameters are learned by minimizing the reconstruction error, which is measured in the squared loss

$$\mathcal{L}_{sq}(\mathbf{W}, \mathbf{b}, \mathbf{b}') = \frac{1}{2}\sum_{i=1}^{n} \|\mathbf{x}_i - \mathbf{x}'_i\|^2. \tag{3}$$

The denoising autoencoder (DA) incorporates a slight modification from the traditional autoencoders. For each input $\mathbf{x}_i$, it picks a fixed percentage of the features uniformly at random and sets them to zero (effectively removing the features), while keeping others untouched. It then maps this corrupted input $\tilde{\mathbf{x}}_i$ into the hidden representations $\tilde{\mathbf{h}}_i$, which is then mapped back as $\tilde{\mathbf{x}}'_i$ to reconstruct the original uncorrupted input, minimizing

$$\mathcal{L}_{sq}(\mathbf{W}, \mathbf{b}, \mathbf{b}') = \frac{1}{2}\sum_{i=1}^{n} \|\mathbf{x}_i - \tilde{\mathbf{x}}'_i\|^2. \tag{4}$$

The stacked denoising autoencoder [23] stacks several DAs together by feeding the hidden representations of the $i^{th}$ DA as input into the $(i+1)^{th}$ DA. The training is performed greedily layer by layer: When the $(i+1)^{th}$ layer is trained, all layers $1...i$ are fixed and noise is only added to the hidden nodes of the $i^{th}$ layer.

Intuitively, by forcing removed features to be reconstructed from the remaining data, the DA learns to convolute features that tend to be correlated. This increases robustness against noise and local transformations, e.g. small translation or rotation. A similar approach has been successfully used for many years in Convolutional Neural Networks (CNN) [15], which leverage the fact that in natural images local pixels are highly correlated with each other — which is hard-coded into the network structure. DAs are more general as they *learn* the convolution patterns, and can be applied to data sets where the feature correlation is unknown (for example in contrast to CNNs, DAs are invariant to arbitrary permutations of the input pixels). However, SdAs suffer from two inherent down-sides: their long training time and sensitivity to several hyper-parameters such as network architecture, learning rates, etc. Carefully tuning these parameters on validation datasets is often very time consuming.

**Pre-training as Feature Generator**. Many researchers have noticed that the pre-training phase in deep learning networks can be seen as some kind of nonlinear feature mappings. For example, [17] showed that the hidden representations computed by either all or partial layers of stacked denoising autoencoders make excellent features for classification with support vector machines. [6] introduced recursively defined kernels which mimic the pre-training of deep feature extractors for the use with support vector machines. Our work follows this line of thinking and shows simpler and more computationally tractable models can also be used for similar purposes.

## 3 Single Layer Construction

Instead of using an autoencoder for pre-training, we propose to reconstruct randomly corrupted data with a linear mapping $\mathbf{W} : \mathcal{R}^d \to \mathcal{R}^d$. To lower the variance, we perform multiple passes over

**Algorithm 1** A single layer construction in MATLAB$^{\text{TM}}$.

```
function [W,h]=lide(X,p);
 X=[X;ones(1,size(X,2))];
 d=size(X,1);
 q=ones(d,1).*p;
 q(end)=1;
 S=X*X';
 Q=S.*(q*q');
 Q(1:d+1:end)=q.*diag(S);
 P=S.*repmat(q,1,d);
 W=((Q+1e-5*eye(d))\P(:,1:end-1))';
```

the training set, essentially corrupting $m$ copies of the original data, and solving for the $\mathbf{W}$ that minimizes the overall squared loss

$$\mathcal{L}_{sq}(\mathbf{W}) = \frac{1}{2} \sum_{j=1}^{m} \sum_{i=1}^{n} \|\mathbf{x}_i - \mathbf{W}\tilde{\mathbf{x}}_{ij}\|^2, \tag{5}$$

where $\tilde{\mathbf{x}}_{ij}$ represents the $j^{th}$ corrupted version of the original input $\mathbf{x}_i$. An input is corrupted by setting each feature randomly to zero with probability $(1 - p)$. To simplify notation we assume that a constant feature is added to the input, $\mathbf{x}_i = [\mathbf{x}_i^\top, 1]^\top$, and an appropriate bias is incorporated within the mapping $\mathbf{W} = [\mathbf{W}, \mathbf{1}]$. The constant feature is never corrupted. For notational simplicity, let us define the $m$-times repetitions of the design matrices $\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_n]$ as $\bar{\mathbf{X}} = [\mathbf{X}, \ldots, \mathbf{X}] \in \mathcal{R}^{d \times nm}$. Let $\tilde{\mathbf{X}}$ be the corrupted equivalent of $\bar{\mathbf{X}}$, *i.e.* $\tilde{\mathbf{X}} = [\tilde{\mathbf{x}}_{1,1}, \ldots, \tilde{\mathbf{x}}_{n,1}, \tilde{\mathbf{x}}_{1,2}, \ldots, \tilde{\mathbf{x}}_{n,m}]$. We can then define two (scaled) outer-product matrices as:

$$\mathbf{Q} = \frac{1}{m}\tilde{\mathbf{X}}\tilde{\mathbf{X}}^\top \text{ and } \mathbf{P} = \frac{1}{m}\bar{\mathbf{X}}\tilde{\mathbf{X}}^\top. \tag{6}$$

Note that only $\mathbf{P}$ is defined over the corrupted and uncorrupted data. We can then express eq. (5) as the well-known closed-form solution for ordinary least squares [4]:

$$\mathbf{W} = \mathbf{P}\mathbf{Q}^{-1}. \tag{7}$$

**Virtual Denoising.** The larger $m$, the more different corruptions we average over. Ideally we would like $m \to \infty$, effectively using infinitely many copies of noisy data to compute the denoising transformation $\mathbf{W}$.

By the weak law of large numbers, the matrices $\mathbf{P}$ and $\mathbf{Q}$, as defined in eq. (6), converge to their expected values as $m$ becomes very large. If we are interested in the limit case, where $m \to \infty$, we can derive a closed-form for these expectations and express the corresponding mapping $\mathbf{W}$ as

$$\mathbf{W} = E[\mathbf{P}]E[\mathbf{Q}]^{-1}. \tag{8}$$

In the remainder of this section, we compute the expectations of $\mathbf{Q}$ and $\mathbf{P}$. For now let us focus on $\mathbf{Q}$, whose expectation is defined as

$$E[\mathbf{Q}] = \sum_{i=1}^{nm} E\left[\tilde{\mathbf{x}}_i\tilde{\mathbf{x}}_i^\top\right]. \tag{9}$$

The off-diagonal entries of $\tilde{\mathbf{x}}_i\tilde{\mathbf{x}}_i^\top$ are uncorrupted if the two features both "survived" the corruption, which happens with probability $p^2$. For the diagonal entries this holds with probability $p$. Let us define a vector $\mathbf{q} = [p, \ldots, p, 1]^\top \in \mathcal{R}^{d+1}$, where $\mathbf{q}_\alpha$ represents the probability of a feature $\alpha$ "surviving" the corruption. As the constant feature is never corrupted, we have $\mathbf{q}_{d+1} = 1$. If we further define the scatter matrix of the original uncorrupted input as $\mathbf{S} = \mathbf{X}\mathbf{X}^\top$, we can express the expectation of the matrix $Q$ as

$$E[\mathbf{Q}]_{\alpha,\beta} = \begin{cases} \mathbf{S}_{\alpha\beta}\mathbf{q}_\alpha\mathbf{q}_\beta & if \quad \alpha \neq \beta \\ \mathbf{S}_{\alpha\beta}\mathbf{q}_\alpha & if \quad \alpha = \beta \end{cases}. \tag{10}$$

By analogous reasoning, we obtain the expectation of $\mathbf{P}$ in closed-form as $E[\mathbf{P}]_{\alpha\beta} = \mathbf{S}_{\alpha\beta}\mathbf{q}_\beta$.

We refer to this linear transformation as Linear Denoiser (LIDE). Algorithm 1 shows a 10-lines MATLAB$^{\text{TM}}$implementation.
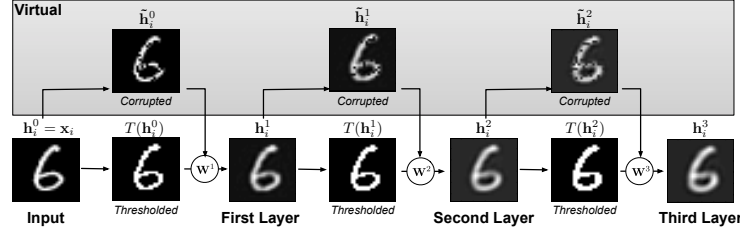
Figure 1: A schematic description of SLIDE. The Corruption is "virtual" in a sense that images are never actually corrupted, as the matrices $\mathbf{W}^k$ can be computed directly in closed-form.

**Algorithm 2** SLIDE in MATLAB$^{\text{TM}}$.

```
function [Ws,hs]=slide(X,p,t,l);
    [d,n]=size(X);
    hs=X;
    Ws=zeros(d,d+1);
    for s=1:l
     Ws(:,:,s)=lide(hs(:,:,s),p);
     hst=double([hs(:,:,s)>t;ones(1,n)]);
     hs(:,:,s+1)=Ws(:,:,s)*hst;
    end;
```

## 4 Stacked Feature Generation

Some of the success of SdAs can be attributed to the fact that they learn *deep* internal representation. So far, LIDE consists of a linear transformation and therefore cannot compete in terms of feature expressiveness. Inspired by the layer-wise stacking of DAs and DBNs, we stack several LIDE layers together by feeding the representations of the $k^{th}$ denoising layer as the input to the $(k + 1)^{th}$ layer. The training is performed greedily layer by layer: When the $(i + 1)^{th}$ layer is trained, all layers $1...l$ are fixed, which means we only learn the $(k + 1)^{th}$ denoising matrix $\mathbf{W}^{k+1}$. For a given input $\mathbf{x}_i$, let $\mathbf{h}_i^k$ denote the output of the $k^{th}$ LIDE transformation. For notational simplicity let us denote $\mathbf{h}_i^0 = \mathbf{x}_i$.

To be able to move beyond a linear transformation, we need to apply a non-linear "squashing"-function between the layers. Several choices might be possible, including sigmoid, or $\tanh$. However in our experiments we simply use a threshold function $T(a) = \delta_{a>t} \in \{0, 1\}$ for some $t$, which we apply element-wise on vectors.

We obtain each layer's representation from the previous layer through the transformation $\mathbf{h}^k = \mathbf{W}^k T(\mathbf{h}^{k-1})$, *i.e.* we threshold the input before the denoising transformation. Analogously to eq. (5), each transformation $\mathbf{W}^{k+1}$ is learned by minimizing the denoising reconstruction error of the previous LIDE output $\mathbf{h}^k$,

$$\mathbf{W}^{k+1} = \underset{\mathbf{W}}{\operatorname{argmin}} \sum_{j=1}^m \sum_{i=1}^n \|\mathbf{h}_i^k - \mathbf{W}\tilde{\mathbf{h}}_{ij}^k\|^2. \tag{11}$$

We solve eq. (11) with the closed form solution for eq. (8) as $m \to \infty$. Figure 1 depicts a schematic layout of the work-flow of SLIDE. Algorithm 2 shows a 9-lines MATLAB$^{\text{TM}}$implementation.

## 5 SVM Training

When the layers of SLIDE are computed, we regard them as features for SVM classification. For simplicity, we use the RBF kernel throughout this paper. The RBF kernel, described in (1), accesses individual inputs only through pairwise distances. However, in each representation, the average distance between data points may vary significantly. Concatenating all the hidden layers and using a single kernel width $\sigma$ across all input features would over-emphasize the impact of some layers over

others. We overcome this problem by introducing a specific kernel width for each layer $\sigma_0, \ldots, \sigma_l > 0$. The kernel function, for the case where the first $l$ hidden layers (and the raw input data) are used, then becomes

$$k(\mathbf{x}_i, \mathbf{x}_j) = \exp - \left( \frac{1}{\sigma^2} \sum_{t=0}^{l} \frac{\|\mathbf{h}_i^t - \mathbf{h}_j^t\|_2^2}{\sigma_t^2} \right). \tag{12}$$

For our setup we set $\sigma = 1$ and learn the individual $\sigma_t$ with the method described in the following subsection.

**Kernel Parameters Learning.** The features from the different layers $\mathbf{h}^0, \ldots, \mathbf{h}^t$ might be of varying utility [18] for the final discrimination task. We suspect that the exact values of $\sigma_t$ might have a noticeable influence on the classification accuracy. As the computation time of cross-validation grows exponentially with the number of parameters, we investigate the benefits of learning the best values for $\sigma_0, \ldots, \sigma_l$ automatically from the data. [5] propose a straight-forward but effective algorithm to learn multiple kernel parameters for support vector machines with gradient descent. We use Chapelle's publicly available code[1] after a trivial modification[2] to learn all $\sigma_t$ in (12). We would like to emphasize that the whole time required for training the SVM *and* optimizing for the individual kernel widths took on the order of minutes even for the larger data sets. As the code by [5] is only implemented for binary settings, we did not apply this technique to the multi-class dataset (MNIST). We also did not use kernel parameter learning for RBFs on raw input, as in this setting cross validation yielded better results. It is important to note that the kernel parameter learning equally applies to both SdA and SLIDE.

## 6 Results

We evaluate our algorithm on several data sets from the deep-learning benchmark collection[3], including: The MNIST handwriting digits recognition dataset[4], the Rectangles-images, the Rectangles and the Convex data set.

**Datasets.** The *MNIST* data consists of 60,000 training and 10,000 testing images of handwritten digits $0 - 9$, where each image is of size $28 \times 28$ pixels. The learning task is to predict the digit identity from the image. The *Rectangles* dataset has $1,200$ training and $50,000$ testing images, also of size $28 \times 28$. Each image contains a rectangle, whose border has a pixel value of $1$ (white), while all other pixels are $0$ (black). The learning task is to determine whether the rectangle has larger width or length. The *Rectangles-Images* dataset is motivated by the same learning task, except that the background and the rectangle are created of two noisy image patches. Also, it has $12,000$ training and $50,000$ testing images. The last dataset in our evaluation is the *Convex* dataset. It consists of $8,000$ training and $50,000$ testing images, of size $28 \times 28$. The images are made of binary pixels (where 0 is black and 1 white). The task is to determine whether all white pixels in a given image form a single convex region.

**Weight Matrices.** Figure 2 visualizes the reconstruction weights for four example pixels on Rectangles, Rectangles-Images, Convex and the MNIST data set at various layers of SLIDE. Each image shows one row of $\mathbf{W}^k$ (without bias weight), where the color reflects the weight value. The figure shows a clear trend towards "fuzzier" reconstruction as the depth increases. In both rectangles data sets (top row) each pixel is reconstructed from neighboring pixels with a tendency towards vertical *or* horizontal offset, thus incorporating the inherent structure from the rectangles in the data. The clarity of this trend is particularly impressive for the Rectangles-Images data set, where the rectangles consist of very noisy image patches. In the convex and MNIST data sets, the pixel is reconstructed from a small circular patch of surrounding pixels. In the case of MNIST, small non-zero weights also exist in distant corner locations, originating from the fact that some pixels may only be non-zero in one or two images in the training data set.

**Experiments Settings.** For all data sets, the training and testing splits are pre-defined. We create an additional random $80/20$ split on the training set for the purpose of parameter tuning.

---

[1]http://olivier.chapelle.cc/ams/

[2]Each iteration we take a gradient-step with respect to all kernel widths instead of just the one global $\sigma$.

[3]http://tinyurl.com/64fgmzv

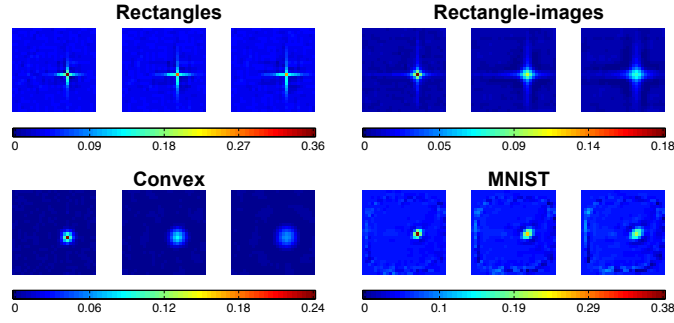[4]http://yann.lecun.com/exdb/mnist/

Figure 2: Reconstruction weights at different layers for the Rectangles, Rectangles-Images, Convex and MNIST data set. The reconstructed pixel itself is most correlated and therefore marked dark red. The cruciform lines in the top row illustrate that the de-noising weights have adapted to the shape of the rectangles. The convex and MNIST data set (bottom row) show a more spherical neighborhood structure, as no obvious neighborhood pattern is inherent in the data.
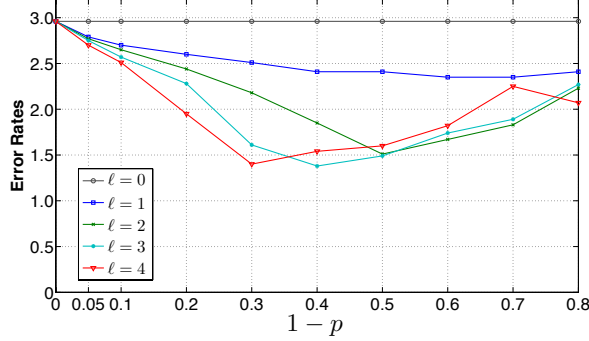


Figure 3: The Rectangles test error as a function of the noise level $(1-p)$ and number of layers $\ell$. The graph shows clearly that deep layers provide a noticeable improvement up to $\ell = 2$ layers.

To find the best combination of noise level $p$ and de-noising layers $\ell$, we run cross-validation on the validation set. As the features across all data sets are naturally within the interval $[0, 1]$, we set the threshold parameter to $t = 0.5$ for all experiments.[5] As described in the previous section, we used kernel parameter learning and SVMs with a RBF kernel [8] for our classification, which we refer to as (SLIDE-SVM). For comparison, we evaluate against three other methods: We use SVMs with a RBF kernel on the raw input (*Raw-SVM*), where we use cross-validation to select the kernel width $\sigma^2$ and the regularization trade-off $C$, as mentioned in the previous section. Further, we use deep neural networks (*SdA*), which were pre-trained with SdA [23] and fine-tuned with back-propagation. We use the network architecture, noise level, and learning rates recommended by the authors of [14] through personal communication. We also use the hidden representations of the SdA as input for the SVM, (*SdA-SVM*), in the exact same fashion as SLIDE-SVM. Finally, other linear transformation are also evaluated, such as PCA, Whitening and random projection (*PCA-SVM, White-SVM, Rand Proj-SVM*). For PCA and Whitening, we reduce the feature dimension by only keeping $p$ percent of variance, and the $p$ is selected by cross-validation.

**Parameter sensitivity.** Figure 3 displays the classification error on the Rectangles data set as a function of the noise level $p$. The four colored lines correspond to different depths $\ell = 0, \ldots, 4$, where $\ell = 0$ or $p = 1$ correspond the original raw input[6] not processed with SLIDE. In general,

---

[5]This choice can be further justified as the squared loss predictor approximates the probability that a binary target has value 1 [4] making $0.5$ the ideal cutoff to minimize the prediction error.

[6]The results for $p = 1$ do not match table 1 because for this graph the kernel parameters were learned, which is sub-optimal for scenarios with few parameters.

| Error in % | Raw-SVM | SdA | SdA-SVM | SLIDE-SVM | Rand Proj-SVM | PCA-SVM | White-SVM |
|---|---|---|---|---|---|---|---|
| **MNIST** | 1.49 | **1.31** | 1.41 | 1.36 | 1.47 | 1.40 | 1.59 |
| **Rect** | 2.45 | 2.43 | 1.10 | 1.38 | 2.94 | 1.54 | **0.96** |
| **Rect-imgs** | 23.29 | 23.07 | 22.70 | **22.39** | 23.27 | 22.55 | 22.58 |
| **Convex** | 18.72 | 17.51 | 17.59 | 12.18 | 18.13 | **11.93** | 18.41 |

Table 1: Error rates (in percent) of deep neural networks with stacked auto-encoder (SdA) pre-training, Support Vector Machines without pre-training (Raw-SVM) and SVMs with SLIDE features (SLIDE SVM).

SLIDE appears to be somewhat sensitive to the exact choice of $p$ and $\ell$, and we choose it by cross validation for all experiments. There is a clear trend that deep layers $\ell > 1$ improve over a single layered transformation $\ell = 1$.

**Experiments Results.** All classification results are shown in table 1. A few general trends can be observed. First, using SLIDE feature pre-processing yields considerable improvements over results with original features (Raw-SVM) on *all* data sets. In fact, transforming the features with SLIDE makes SVM outperform even Deep Neural Nets (SdA) in all but one (MNIST) learning tasks. Finally SLIDE even outperforms the much more complex SdA features on two data sets (Rect-imgs,Convex) and obtains equivalent results (up to significance) on MNIST. Especially on Convex and the Rect-Images data sets, SLIDE clearly outperforms all other algorithms. Results from PCA (PCA-SVM) are also significantly better than original features (Raw-SVM), and is the best in the Convex dataset, but worse than SLIDE on all others. Whitening (White-SVM) tops the Rectangle dataset, but does not yield significant improvements on other datasets. These results are very encouraging as SLIDE is trivial to implement and orders of magnitudes faster than SdA.

**Running Time.** Table 2 compares the running times for feature generation with SdA and SLIDE. All timings are performed on a desktop with dual Intel[TM] Six Core Xeon X5650 2.66GHz processors. To train the SdA we use the highly optimized Theano open-source package[7] which is carefully parallelized and in our experiments utilizes a state-of-the-art GPU[8] with additional 64 cores. No explicit parallelization was used for SLIDE. The results show a three orders of magnitude speed-up across all data sets, reducing the pre-training time from several hours to a few seconds.

| Time | MNIST | Rect | Rect-imgs | Convex |
|---|---|---|---|---|
| SdA | 8h 24m | 1h 30m | 2h 40m | 6h 50m |
| SLIDE | 22s | 5s | 8s | 6s |
| $\times$ speedup | 1377 | 1080 | 1207 | 4100 |

Table 2: Running time required for feature generation with SdA and SLIDE.

# 7   Related Work

Related work can be categorized into three lightly correlated dimensions: layer-wised unsupervised pre-training, learning from partially corrupted data, and linking SVMs with neural networks. In the area of layer-wised unsupervised pre-training, [11], [17], [2], [16] and [21] provide various successful approaches. They all propose to pre-train the neural network by some unsupervised training criterion as an initialization for back-propagation or as features for other algorithms (e.g. SVM). In contrast, our method does not require any training through gradient descent and is orders of magnitudes faster.

---

[7]http://deeplearning.net/software/theano/
[8]NVIDIA Quadro FX 1800 768MB GDDR3

[20] use carefully constructed random projections to create features that approximate kernelization for linear SVMs. In the area of learning from partially corrupted data, [23] proposes using Stacked Denoising Auto-encoder(SdA) to reconstruct the corrupted data, and demonstrates that the learnt representations are robuster. Our method is heavily inspired by their work and can be viewed as a convex closed-form transformation that mimics their feature generator.

[6] also investigate linking neural networks with SVMs through deep kernels. In their work, they construct new recursively composed positive semi-definite kernel functions which can be viewed as mimicking the layer by layer training of neural networks. Different from our work, their method is not based on feature de-noising and still requires computationally expensive "fine-tuning" through distance metric learning [24].

# 8 Conclusion

We introduced SLIDE, a novel algorithm based on stacked linear denoising autoencoders for extremely fast layer-wise deep pre-training for feature generation. We derived a simple closed-form solution that can be implemented in a few lines of MATLAB$^{\text{TM}}$. We demonstrated that SLIDE for SVM classification can match (or out-perform) the classification results of SdA features – on three out of four data set even beyond the low error rates of deep neural networks. Most notably, the running time of SLIDE is reliably in the order of seconds even on data sets with tens of thousands of data points.

As future directions we plan to investigate other classifiers and different learning settings. As SLIDE is entirely unsupervised, it lends itself naturally towards semi-supervised and transfer learning tasks.

Because SLIDE is so straightforward to implement, only takes seconds to compute and improves results for SVM classification with surprising consistency, we have great hopes that it will find use as a general pre-processing algorithm across many areas of machine learning.

# References

[1] Y. Bengio. Speeding up stochastic gradient descent. In *Neural Information Processing Systems Workshop on Efficient Machine Learning*, 2007.

[2] Y. Bengio, P. Lamblin, D. Popovici, H. Larochelle, U. D. Montral, and M. Qubec. Greedy layer-wise training of deep networks. In *In NIPS*. MIT Press, 2007.

[3] Y. Bengio and Y. LeCun. Scaling learning algorithms towards AI. *Large-Scale Kernel Machines*, 34, 2007.

[4] C. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[5] O. Chapelle, V. Vapnik, O. Bousquet, and S. Mukherjee. Choosing multiple parameters for support vector machines. *Machine Learning*, 46(1):131–159, 2002.

[6] Y. Cho and L. Saul. Large-margin classification in infinite neural networks. *Neural Computation*, 22(10):2678–2697, 2010.

[7] R. Collobert and J. Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM, 2008.

[8] C. Cortes and V. Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[9] D. Erhan, Y. Bengio, A. Courville, P. Manzagol, P. Vincent, and S. Bengio. Why Does Unsupervised Pre-training Help Deep Learning? *The Journal of Machine Learning Research*, 11:625–660, 2010.

[10] G. Hinton, S. Osindero, and Y. Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[11] G. E. Hinton and S. Osindero. A fast learning algorithm for deep belief nets. *Neural Computation*, 18:2006, 2006.

[12] G. E. Hinton and R. R. Salakhutdinov. Reducing the Dimensionality of Data with Neural Networks. *Science*, 313(5786):504–507, 2006.

[13] H. Larochelle, Y. Bengio, J. Louradour, and P. Lamblin. Exploring strategies for training deep neural networks. *The Journal of Machine Learning Research*, 10:1–40, 2009.

[14] H. Larochelle, D. Erhan, A. Courville, J. Bergstra, and Y. Bengio. An empirical evaluation of deep architectures on problems with many factors of variation. In *In ICML*, 2007.

[15] Y. LeCun and Y. Bengio. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, pages 255–258, 1995.

[16] H. Lee, C. Ekanadham, and A. Ng. Sparse deep belief net model for visual area V2. *Advances in neural information processing systems*, 20, 2007.

[17] H. Lee, Y. Largman, P. Pham, and A. Y. Ng. Unsupervised feature learning for audio classification using convolutional deep belief networks. In *Advances in Neural Information Processing Systems 22*, pages 1096–1104. 2009.

[18] G. Montavon, M. L. Braun, and K.-R. Müller. Layer-wise analysis of deep networks with gaussian kernels. In J. Lafferty, C. K. I. Williams, R. Zemel, and J. Shawe-Taylor, editors, *Advances in Neural Information Processing Systems 23*. 2010.

[19] V. Nair and G. Hinton. 3d object recognition with deep belief nets. In Y. Bengio, D. Schuurmans, J. Lafferty, C. K. I. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems 22*, pages 1339–1347. 2009.

[20] A. Rahimi and B. Recht. Random features for large-scale kernel machines. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 1177–1184. MIT Press, Cambridge, MA, 2008.

[21] M. Ranzato, L. Boureau, and Y. LeCun. Sparse feature learning for deep belief networks. *Advances in neural information processing systems*, 2007.

[22] B. Schölkopf and A. J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, Cambridge, MA, 2002.

[23] P. Vincent, H. Larochelle, Y. Bengio, and P. Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103. ACM, 2008.

[24] K. Weinberger and L. Saul. Distance metric learning for large margin nearest neighbor classification. *The Journal of Machine Learning Research*, 10:207–244, 2009.